Parallel Software/Hardware

Chapter 2.1 & 2.2

Spring 2017

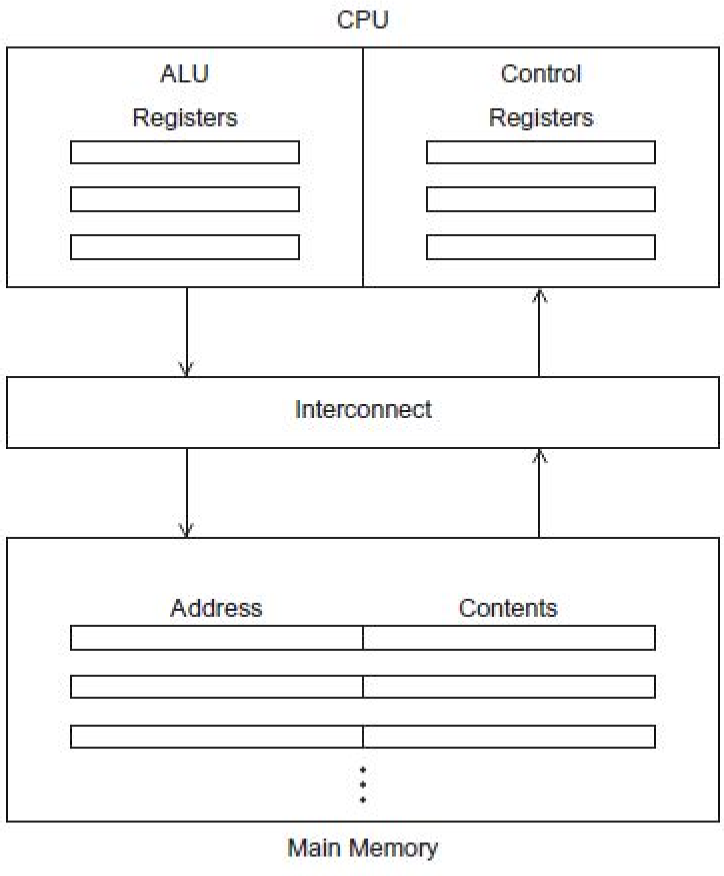
Re-cap

- Parallel algorithms are key as our single processors quit accelerating and multi-core wins the day.
- Evaluate parallel algorithms using:
  - Speedup – $S = T_s(n) / T_p(n)$
  - Efficiency – $E = S / p$
- Amdahl's law limits speedup by amount of serial code
  - Gustafson's law says just make problem bigger
- Types of Parallelism
  - Data
  - Task
- Types of Parallel Systems
  - Shared Memory
  - Distributed Memory

John von Neumann

- Hungarian-American mathematician (1903-1957).
- Contributions to mathematics, economics, computer science, and statistics.
- Member of Manhattan Project and Institute for Advanced Study.
- Proposed a design for a digital computer (EDVAC) in 1945 that later became the von Neumann model.
- Introduced cellular automata.
- Designed merge sort algorithm.

The Von Neumann Model (1945)

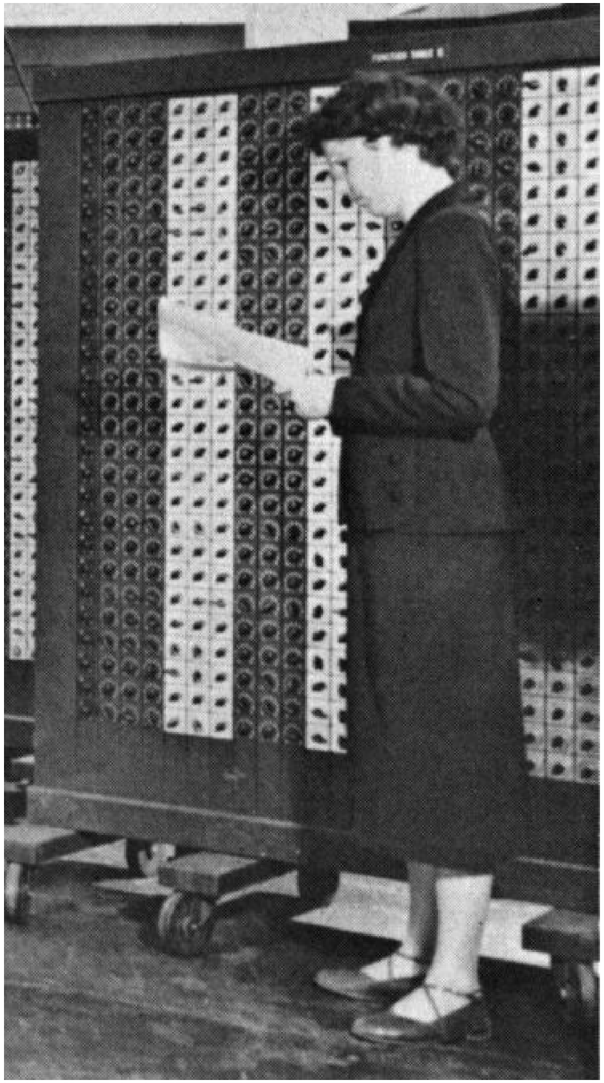


- Control unit – responsible for deciding which instruction should execute. (the boss)
- ALU (Arithmetic and logic unit) – responsible for actually doing the work. (the worker)
- Memory:
  - A collection of location to store both data and instructions.
  - Each location has an address.

Betty Holberton

- Programmer on ENIAC
- Invented mainframe Sort/Merge
- Statistics package for 1950 census

"solved more problems in her sleep than other people did awake"



Process (task)

- An instance of a computer program being executed.
  - Memory
    - Program
    - Data
  - Security (who, what where)
  - State
    - Register Values
    - Program counters
    - Resources (file handler, sockets)

Multitasking

- Gives the illusion that a single processor system is running multiple programs simultaneously.
- Each process takes turns running.
  (**time slice**)
- After its time is up, it waits until it has a turn again and **context switches**.

Threading

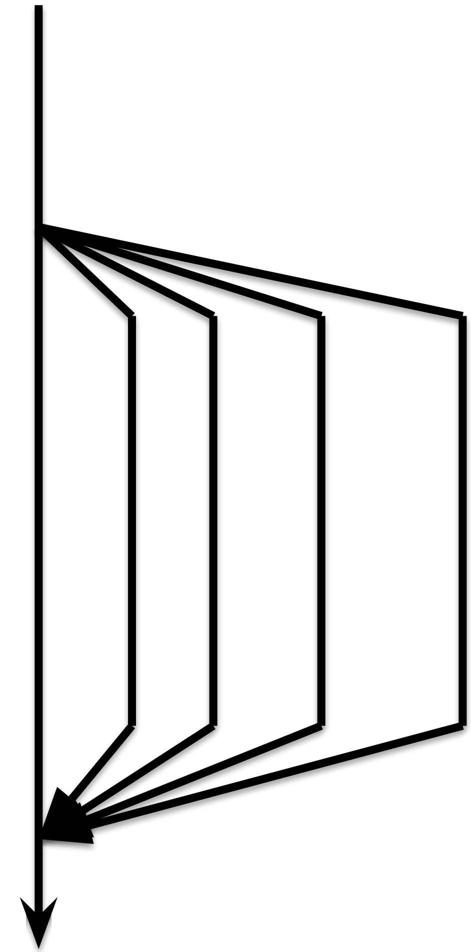
- Similar to multitasking but within a single process.
- Originally a method for hiding memory latency.
- Example, covert dot-product of two vectors x,y of length n and covert to a 4 “threaded” environment

```
dp = 0;
for (i=0; i < n; i++) {
  dp += x[i] * y[i]
}
```

```
dp = 0;
for (int k=0; k < 4; k++) {
  partialProd(k, k*n/4, n/4);
}
for (int i=0; i < 4; i++) {
  dp += pdp[i];
}
void partialProd(int k, int a, int b) {
  pdp[k] = 0;
  for (i=a; i<a+b; i++) {
    pdp[k] += x[i]*y[i];
  }
}
```

Create Threads

```
dp = 0;

for (int k=0; k < 4; k++) {
  createThread(partialProd(k, k*n/4, n/4));
}

waitOnThreadsToComplete();

for (int i=0; i < 4; i++) {
  dp += pdp[i];
}

void partialProd(int k, int a, int b) {
  pdp[k] = 0;
  for (i=a; i<a+b; i++) {
    pdp[k] += x[i]*y[i];
  }
}
```



Thread = Lightweight Process

- Threads within the same process
- Share the memory address space
- Each has its own registers, program counter and stack pointer
- The OS schedules processes but a thread library function schedules threads within a process
  - Windows/Solaris are slightly different
    - Versions of linux also know about threads ☺
- Kernel threads are special

Terms

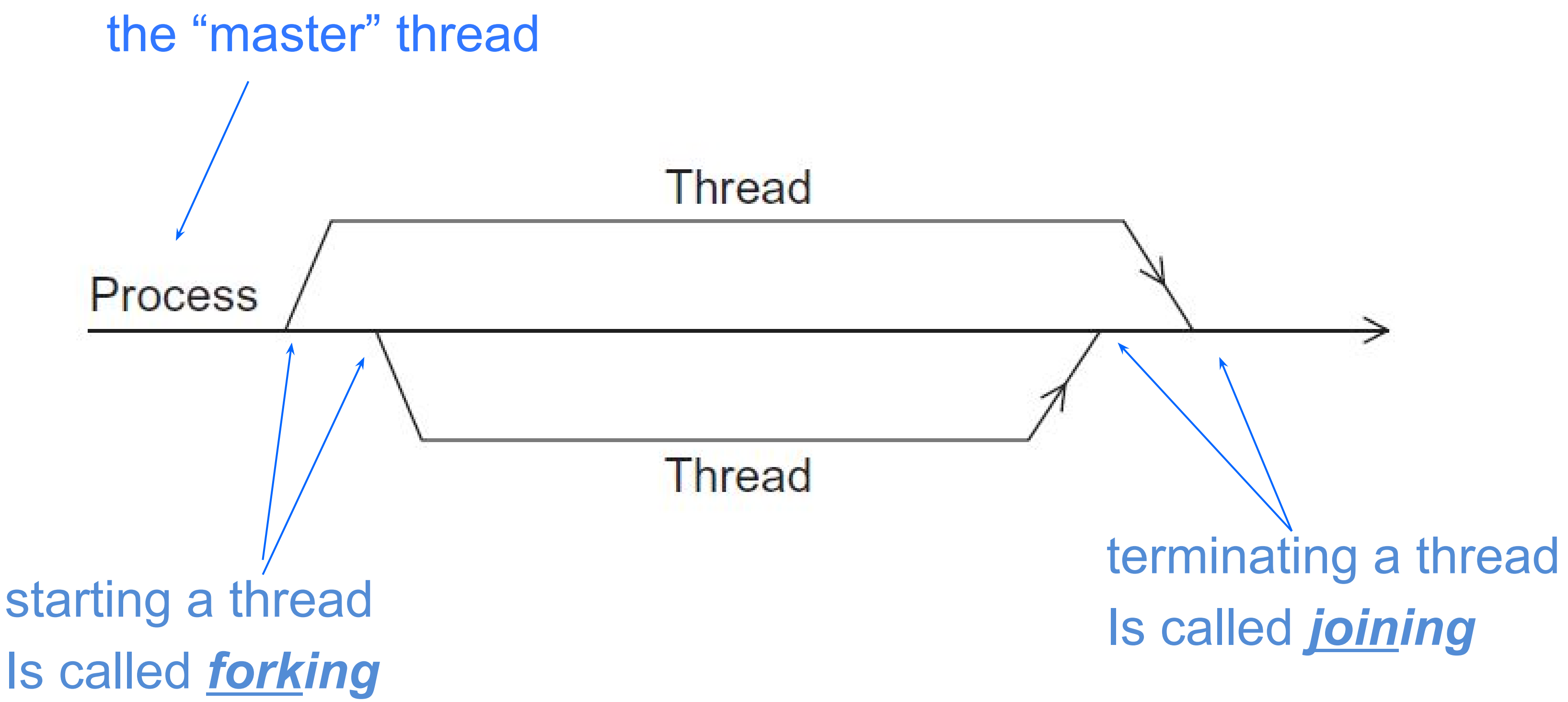


Figure 2.2

The Memory Wall

- Memory is often the bottleneck
- Memory performance measured in
  - Latency – “delay”
  - Bandwidth – “trunk size”

Memory Wall (example)

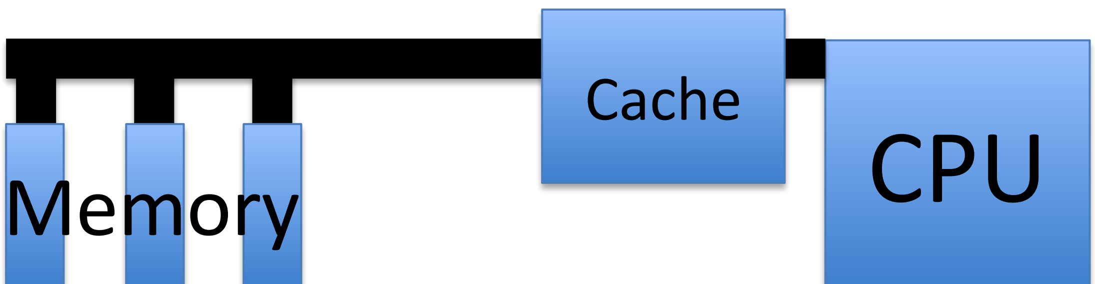
- 1 GHz processor (1 ns clock) with a multiply-add units capable of executing a multiply/add in each cycle
- The peak processor rating is 2 GFLOPS
  …. 2 * 10^9 floating point operation per second!
- We want to compute the dot product of two vectors

$$DP = \sum_{i=0}^{n} x[i] * y[i]$$

- Need two operands every ns for peak processor performance
- If DRAM has a latency of 100 ns, then it can supply only one operand every 100 cycle. Hence, can do a multiply/add (2 FLOPS) every 200 ns. That is one FLOP every 100 ns.
- Actual performance = 10 MFLOPS

Overcoming Memory Wall

- Cache: take advantage of spatial and temporal locality.



- Prefetching: Improve performance by pre-fetching. In this case, performance depends on memory bandwidth and not its latency. In the example above, if bandwidth is such that one operands can be fetched every 5ns, then can do a multiply/add every 10 ns. That is one FLOP every 5ns (200 MFLOP) – still not enough to match the 2GFLOP processor.
- Multithreading – Do useful things while waiting.

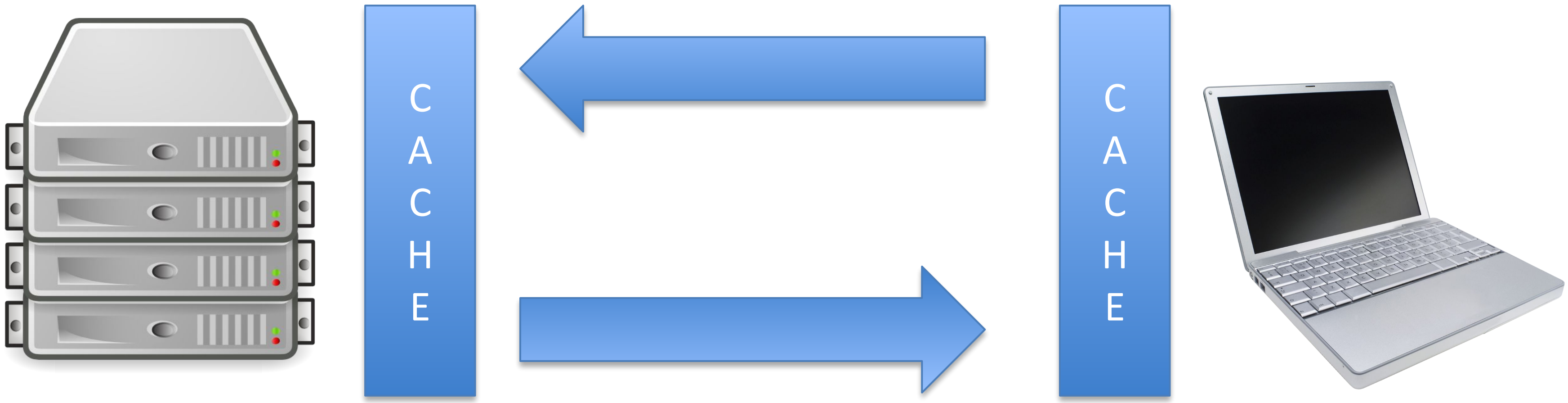
Principle of locality

- Accessing one location is followed by an access of a nearby location.
- **Spatial locality** – accessing a nearby location.
- **Temporal locality** – accessing in the near future.

Principle of locality

```
float z[1000];
…
sum = 0.0;
for (i = 0; i < 1000; i++)
   sum += z[i];
```

Web Cache



Locating Cached Data (Go Fish)

- Questions:
  - Is my data in the cache?
  - How do I find it?
  - What if it is not in the cache?
- Two broad solutions:
  - Cache items can go anywhere in cache (associative)
    - Must have some method of looking up
  - Each memory location in certain locations (direct)
    - Must have “rule” of mapping

Cache mappings

- **Full associative** – a new line can be placed at any location in the cache.
- **Direct mapped** – each cache line has a unique location in the cache to which it will be assigned.
- ***n*-way set associative** – each cache line can be place in one of *n* different locations in the cache.

Cache Eviction

- When more than one line in memory can be mapped to several different locations in cache we also need to be able to decide which line should be replaced or evicted.
  - First in First Out (FIFO)
  - Least Recently Used (LRU)
  - Least Frequently Used (LFU)

Example

Memory Index	Cache Location		
	Fully Assoc	Direct Mapped	2-way
0	0, 1, 2, or 3	0	0 or 1
1	0, 1, 2, or 3	1	2 or 3
2	0, 1, 2, or 3	2	0 or 1
3	0, 1, 2, or 3	3	2 or 3
4	0, 1, 2, or 3	0	0 or 1
5	0, 1, 2, or 3	1	2 or 3
6	0, 1, 2, or 3	2	0 or 1
7	0, 1, 2, or 3	3	2 or 3
8	0, 1, 2, or 3	0	0 or 1
9	0, 1, 2, or 3	1	2 or 3
10	0, 1, 2, or 3	2	0 or 1
11	0, 1, 2, or 3	3	2 or 3
12	0, 1, 2, or 3	0	0 or 1
13	0, 1, 2, or 3	1	2 or 3
14	0, 1, 2, or 3	2	0 or 1
15	0, 1, 2, or 3	3	2 or 3

Table 2.1: Assignments of a 16-line main memory to a 4-line cache

Cache lines

```
double A[MAX][MAX], x[MAX], y[MAX];
. . .
/* Initialize A and x, assign y = 0 */
. . .
/* First pair of loops */
for (i = 0; i < MAX; i++)
   for (j = 0; j < MAX; j++)
      y[i] += A[i][j]*x[j];
. . .
/* Assign y = 0 */
. . .
/* Second pair of loops */
for (j = 0; j < MAX; j++)
   for (i = 0; i < MAX; i++)
      y[i] += A[i][j]*x[j];
```

Cache Line	Elements of A			
0	A[0][0]	A[0][1]	A[0][2]	A[0][3]
1	A[1][0]	A[1][1]	A[1][2]	A[1][3]
2	A[2][0]	A[2][1]	A[2][2]	A[2][3]
3	A[3][0]	A[3][1]	A[3][2]	A[3][3]

Cache Line – Store more than just a single address, instead we store x addresses "line of data".

Data Layout and Cache Lines

```
for (j = 0; j < 1000; j++)
  column_sum[j] = 0.0;
  for (i = 0; i < 1000; i++)
    column_sum[j] += b[i][j];
```

- The code fragment sums columns of the matrix b into a vector column_sum.
- The vector column_sum is small and easily fits into the cache
- The matrix b is accessed in a column order
- With row major storage, strided access results in very poor performance.

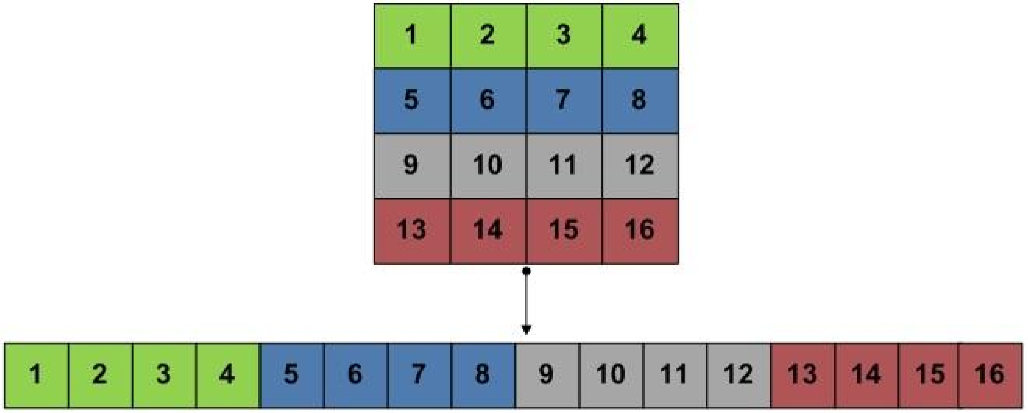
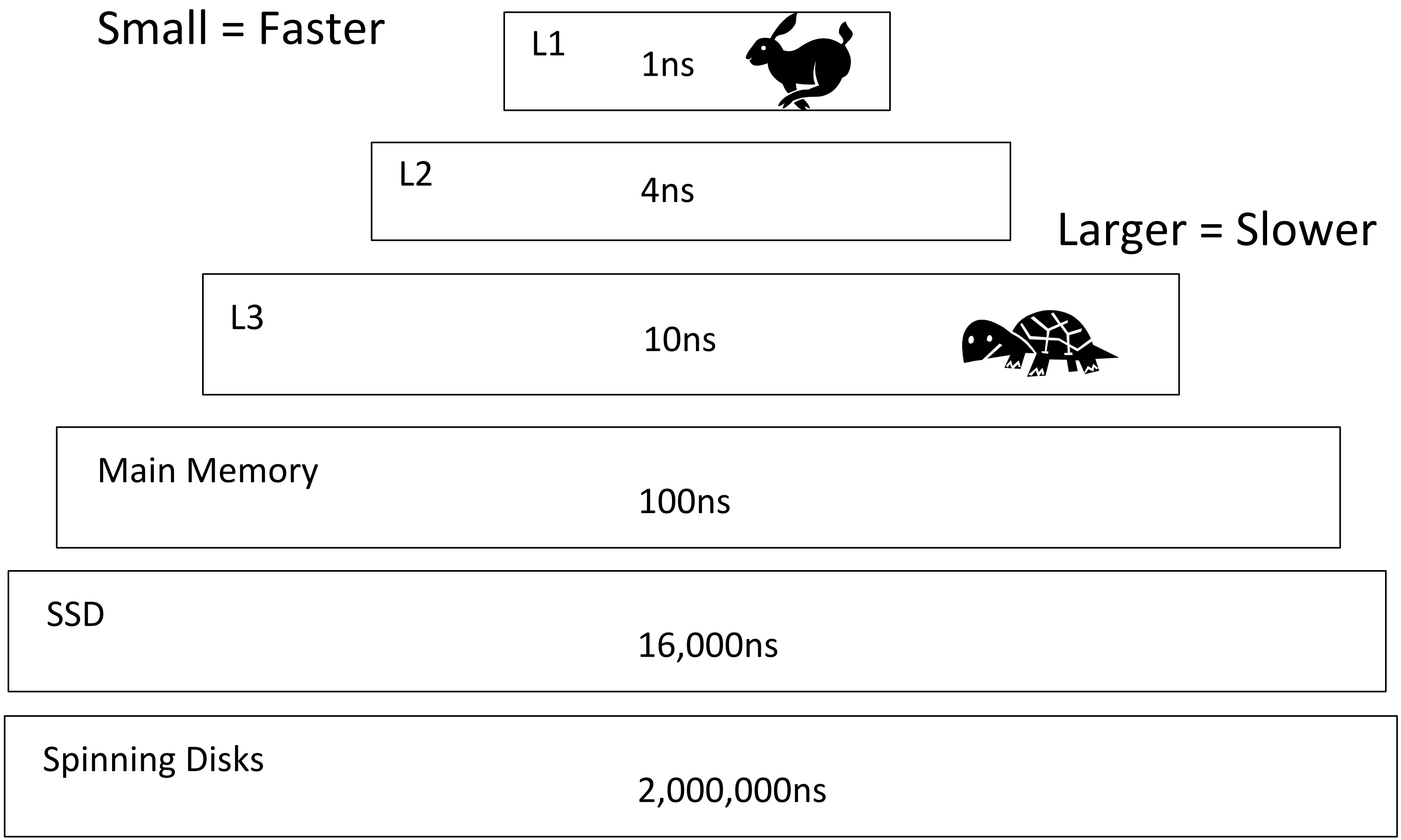


Image Source Wikipedia

Writing to Cache

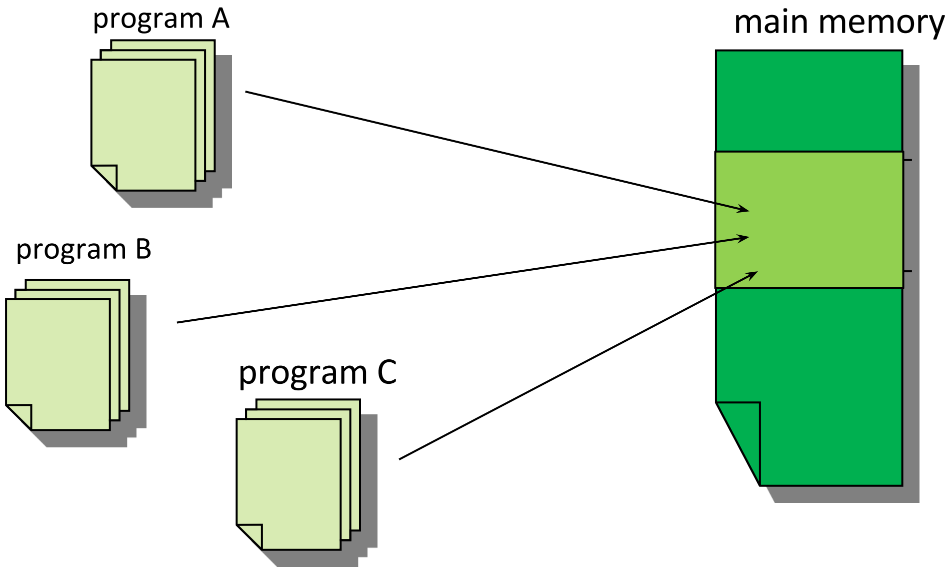
- When a CPU writes data to cache, the value in cache may be inconsistent with the value in main memory.
- **Write-through** caches handle this by updating the data in main memory at the time it is written to cache.
- **Write-back** caches mark data in the cache as dirty. When the cache line is replaced by a new cache line from memory, the dirty line is written to memory.

Levels of Data Access



Virtual memory

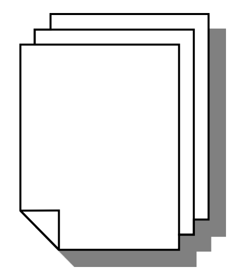
- If we run a very large program or a program that accesses very large data sets, all of the instructions and data may not fit into main memory.
- Virtual memory functions as a cache for secondary storage.
- It exploits the principle of spatial and temporal locality.
- It only keeps the active parts of running programs in main memory.



Virtual memory

- **Swap space** - those parts that are idle are kept in a block of secondary storage.

- **Pages** – blocks of data and instructions.
  – Usually these are relatively large.
  – Most systems have a fixed page size that currently ranges from 4 to 16 kilobytes.



Virtual page numbers

- When a program is compiled its pages are assigned ***virtual*** page numbers.
- When the program is run, a table is created that maps the virtual page numbers to physical addresses.
- A **page table** is used to translate the virtual address into a physical address.

Virtual Address									
Virtual Page Number					Byte Offset				
31	30	···	13	12	11	10	···	1	0
1	0	···	1	1	0	0	···	1	1

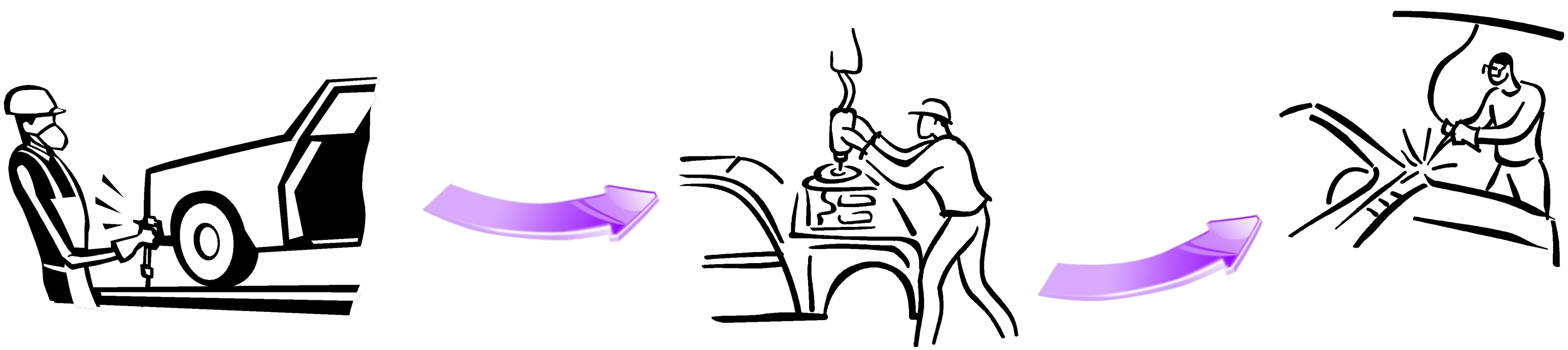
Virtual Address Divided into Virtual Page Number and Byte Offset

Translation-lookaside buffer (TLB)

- Using a page table has the potential to significantly increase each program's overall run-time
- A special address translation cache in the processor.
- It caches a small number of entries (typically 16–512) from the page table in very fast memory.
- **Page fault** – attempting to access a valid physical address for a page in the page table but the page is only stored on disk.

Instruction Level Parallelism (ILP)

- Attempts to improve processor performance by having multiple processor components or **functional units** simultaneously executing instructions.
- **Pipelining** - functional units are arranged in stages.
- **Multiple issue** - multiple instructions can be simultaneously initiated.



Pipelining example

Add the floating point numbers
9.87×10^4 and 6.54×10^3

Time	Operation	Operand 1	Operand 2	Result
1	Fetch operands	9.87×10^4	6.54×10^3	
2	Compare exponents	9.87×10^4	6.54×10^3	
3	Shift one operand	9.87×10^4	0.654×10^4	
4	Add	9.87×10^4	0.654×10^4	10.524×10^4
5	Normalize result	9.87×10^4	0.654×10^4	1.0524×10^5
6	Round result	9.87×10^4	0.654×10^4	1.05×10^5
7	Store result	9.87×10^4	0.654×10^4	1.05×10^5

Pipelining example

```
float x[1000], y[1000], z[1000];

for (int i=0; i < 1000; i++)
    z[i] = x[i] + y[i]
```

- Assume each operation takes one nanosecond.
  - 7 operations per addition.
  - This for loop takes about 7000 nanoseconds.

Pipelining

- Divide the floating point adder into 7 separate pieces of hardware or functional units.
- First unit fetches two operands, second unit compares exponents, etc.
- Output of one functional unit is input to the next.

Time	Fetch	Compare	Shift	Add	Normalize	Round	Store
0	0						
1	1	0					
2	2	1	0				
3	3	2	1	0			
4	4	3	2	1	0		
5	5	4	3	2	1	0	
6	6	5	4	3	2	1	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
999	999	998	997	996	995	994	993
1000		999	998	997	996	995	994
1001			999	998	997	996	995
1002				999	998	997	996
1003					999	998	997
1004						999	998
1005							999

Numbers in the table are subscripts of operands/results.

Pipelining example

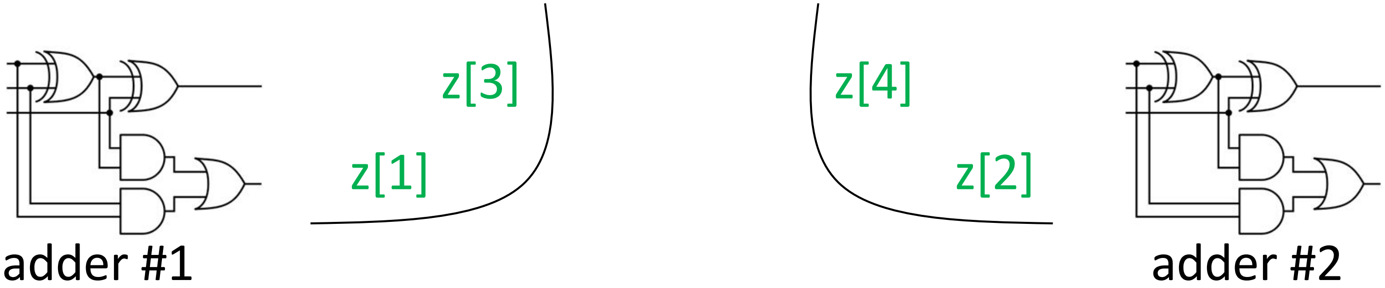
```
float x[1000], y[1000], z[1000];

for (int i=0; I < 1000; i++)
    z[i] = x[i] + y[i]
```

- Each operation still takes one nanosecond.
  - 7 operations per addition (still 7ns per addition)
- How long will 1000 operations take?
  - **1006ns with pipelining!**

Multiple Issue

- Multiple issue processors replicate functional units and try to simultaneously execute different instructions in a program.

```
for (i = 0; i < 1000; i++)
     z[i] = x[i] + y[i];
```



Multiple Issue

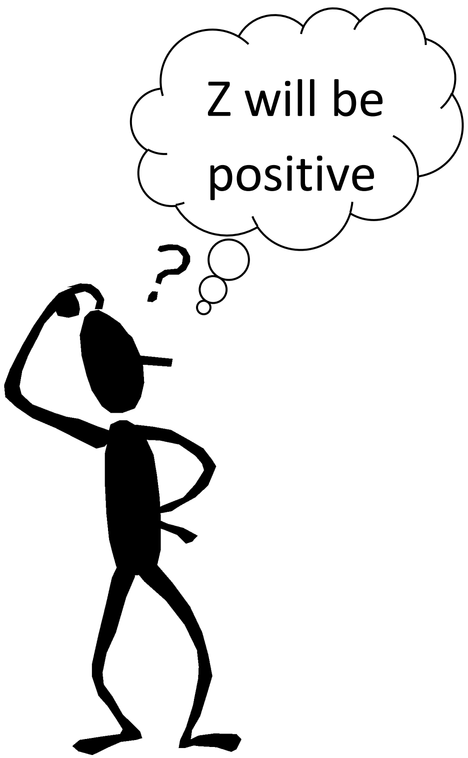
- **VLIW – Very long instruction word -** functional units are scheduled at compile time (static scheduling).

- **Superscaler -** functional units are scheduled at run-time (dynamic scheduling).

Speculation

- In order to make use of multiple issue, the system must find instructions that can be executed simultaneously.
- In speculation, the compiler or the processor makes a guess about an instruction, and then executes the instruction on the basis of the guess.

```
z = x + y;
if ( z > 0)
     w = x;
else
     w = y;
```



If the system speculates incorrectly,
it must go back and recalculate w = y.

Multi-threading

Provides a means to continue doing useful work when the currently executing task has stalled (ex. wait for long memory latency).

Lighter weight than multi-tasking because context switching is usually more costly than thread-switching.*

- Software-based multi-threading (Posix Threads)
  - Hardware still traps on long-latency processes
  - Software handles thread context switch
  - Issues with overhead and “multi-level” control
- Hardware-based multi-threading
  - User defines threads (or kernel)
  - Hardware “helps” in context switch
  - Ex: IBM Power5, Pentium-4

Hardware Multi-threading

- Provides a means to continue doing useful work when the currently executing task has stalled (ex. wait for long memory latency)
- **Fine-grain multithreading**
  - Switch threads after each cycle
  - Interleave instruction execution
  - If one thread stalls, others are executed
- **Coarse-grain multithreading**
  - Only switch on long stall (e.g., L2-cache miss)
  - Simplifies hardware, but doesn’t hide short stalls (such as the stalls resulting from data hazards)
- **SMT** in multiple-issue dynamically scheduled processor
  - Schedule instructions from multiple threads
  - Instructions from independent threads execute when function units are available

Summary

- Von Neumann model
  - Memory bottleneck and overcoming it
- Process vs Threads
- Cache
- Virtual Memory
- Pipelining
- Multi-threading